CSE 335
Winter 2017 Homework 6

Suggested reading:
    EOPL: 2.4 (refresh your memory on define-datatype)
    EOPL: B.1-B.3 (about sllgen)
    EOPL: 3.1-3.2 (implementation of language)

For this homework you will be implementing your first interpreter. To accomplish this, you are using sllgen described in your textbook and discussed in class. Given grammatical and lexical specifications, sllgen will generate a parser that given a program string, it will generate the appropriate Abstract Syntax Tree (AST). You will need to deal with AST in your implementations. So when in doubt, run (show-data-types) boiler plate code to check it.

If you read the BNF grammar you will notice that the already well known up, down, left, right steps reappear. The reason is that we will be implementing a programming language that described the movement of an entity in a 2D grid.

As a rule of thumb, every expression in our language will return a value. Currently the only values that can be returned by the language are described by the "expressed-val" datatype found in the "hw06-env-values.rkt" file. Here onward, these values will be referred to as the "expressed values" of our language.

Also, please read the answer-sheet carefully, it contains several tips.

==========

Given the language described by the following BNF grammar:

```
<program> ::=
            <expr> <expr>* “a-program”

<expr> ::=
           number           "num-expr"
        | up(<expr>)        "up-expr"
        | down(<expr>)      "down-expr"
        | left(<expr>)      "left-expr"
        | right(<expr>)     "right-expr"

        | (<expr> <expr>)   "point-expr"
        | + <expr> <expr>   "add-expr"
        | origin? (<expr>)  "origin-expr"
        | if (<expr>)
           then <expr>
           else <expr>       "if-expr"

        | move (<expr> <expr> <expr>*)  "move-expr"
```

note:
  * represents 0 or more of the thing that is marked.

Semantics:
  - up, down, left, right (<expr>):
    <expr> here will always have to evaluate to a num-expr

  - point-expr (<expr> <expr>): both <expr>'s here will have
    to be num-expr

  - origin? (<expr>): <expr> here is a point-expr

  - if (<expr>) …:   <expr> here will have to be an origin-expr

  - + <expr> <expr>, you can only add together:
    left-expr && left-expr
    left-expr && right-expr
    right-expr && right-expr
    right-expr && left-expr
    down-expr && down-expr
    down-expr && up-expr
    up-expr && down-expr
    up-expr && up-expr

  - move (<expr> <expr> <expr>*):
    the first <expr> has to be a <point-expr>
    <expr> <expr>*, have to be up, down, left, right expressions

==========
1. [10p]
Look in your answer sheet for "the-lexical-spec". Please describe
in your own words how the program strings are "chopped up" based
on this specification.

==========
2. [20p]
Write the grammar specification for the above described language.
(We practiced a bit on this in class today, for more details
please refer to hw06-answer-sheet.rkt)
==========

3. [40p]
Implement the above language.
==========

As you can notice the above version of the language does not have
a concept of an identifier. Consider these new additions to the
grammar of the language:

```
<expr> ::=
           <initial-language>
         | identifier              "iden-expr"
         | { <var-expr>* <expr> *} "block-expr"

<var-expr> ::=
               | val identifier = <expr>         "val"
               | final val identifier = <expr>   "final-val"
```

We had to make the distinction between <var-expr> and <expr> in
order to be able to write the <block-expr> as is described.

Semantics

- whenever an identifier is encountered, it is evaluated to its bound value (YES, your knowledge of environment data type will help a lot! In fact, we provided the solution code for hw05 that is specifically related to environment data type in hw06-env-values.rkt file. It also contains data type information for values - please read hw06-env-values.rkt carefully.)

- block-expr, can have any number of variable declarations, i.e. <var-expr> followed by any number of expressions <expr>. The value returned by the block-expr, just like in racket, is the value of the last statement. ALL previous expressions still have to be evaluated.

- val identifier = <expr>, it will create a variable with the name "identifier" that is bound to the value returned by the <expr>

- final val, similar to val, but the variable's definition cannot be overridden. This is similar to what you implemented in homework05, problem 2.b.

- the return value of a val, final-val or var-expr are undefined. i.e. you can have them return whatever you wish.

==========

4. [30p]
Implement the language features that support identifiers.

Note:
To solve this last part you will have to make use of the environment datatype (already provided in the hw06-env-values.rkt file). This will require a few changes to the structure of your code so I strongly

recommend that after solving problem 3 you backup that implementation and start implementing these new features on a copy.

SUGGESTIONS:
If your code for problem 4 has problems to pass all the test cases, then it might be easier for you to submit two separate files, one containing the untarnished,(hopefully, correct) solution for problems 1 ~ 3, and another file containing your solution for problem 4. If all language features are implemented correctly, please submit only one file.